Section 22
Schedule Database Function

The *Schedule Database (SDB)* function provides airline schedule data to the ETMS system in the following ways:

- It sends packets of scheduled airline flights (scheduled flight messages [FS]), in a format similar to flight plan messages (FZ), to the *Parser* via *nas.dist* on a continual basis, to be included in *today's* data.
- In response to requests from the *listserver* (see Section 17), it sends lists of flights to or from specific airports for a requested time range.

In addition, commands can be sent to the *SDB* function to enter changes into the database.

NOTE: The *listserver* is sometimes called the *Request Server*, and should not be confused with the *SDB List Server*.

The schedule database (SDB) consists of a main table of airline flight schedules stored in departure time order and (within each minute of departure time) in flight ID order. A flight schedule consists of the following:

- Flight ID
- Scheduled departure time
- Scheduled arrival time
- Departure airport
- Arrival airport
- Estimated time en route
- Aircraft type
- Days of the week on which the flight occurs
- Dates on which the flight becomes effective/discontinues
- Dates on which the flight becomes inhibited/activated
- Indications of flight status (inhibited, canceled)

The departure time sort of the database allows quick access to multiple schedules in the order in which they are to be sent to the *NAS Distributor*. There are a number of index files accompanying the main file, which allow rapid access to flight schedules using the following keys:

- Departure time (in 15-minute buckets)
- Arrival airports
- Departure airports
- Flight ID
- Air carrier
- Canceled flights
- Added flights
- Inhibited flights

The departure time index speeds access further by allowing the program to jump into the database at a point within 15 minutes of departure time without having to search from the beginning.

The arrival airports and departure airports indexes are used by the *SDB List Server* (see Section 22.3) to create flight lists (by airport and time period) to send to the *listserver*. These indexes are sorted by one-hour time buckets within each airport name. This provides quick access to each airport's flights during a specific time period, allowing rapid response to **LIST** requests.

The flight ID index is used by the *Process Requests* module of the *Update/Request Server* to locate specific flights in the database to make requested changes to their schedules (see Section 22.2.2).

The air carrier index is used by the *Process Requests* module to make changes for all flights of a particular air carrier- Inhibit (**INHB**) and Activate (**ACTV**) requests.

The canceled flights, added flights, and inhibited flights indexes keep track of flights affected by requests that resulted in changes to the database. The indexes allow quick access to those flights for database updating and maintenance.

Processing Overview

Processing is begun by the *SDB Server* (see Section 22.1). As part of its initialization, the *SDB Server* process connects to the *ETMS Communications* functions. Schedule data and change requests are received through these connections. The process also acts as a TCP/IP server for communication with the *Update/Request Server* (see Section 22.2) and the *SDB List Server* (see Section 22.3) processes. The *Update/Request Server* and the *SDB List Server* each open a channel to the *SDB Server* in their respective initialization routines (see Figure 22-1).

When the *SDB Server* receives a message from the *ETMS Communications* functions, it determines the type of message from the first word or two. The process then passes the message to the *Update/Request Server* or *SDB List Server*, as appropriate.

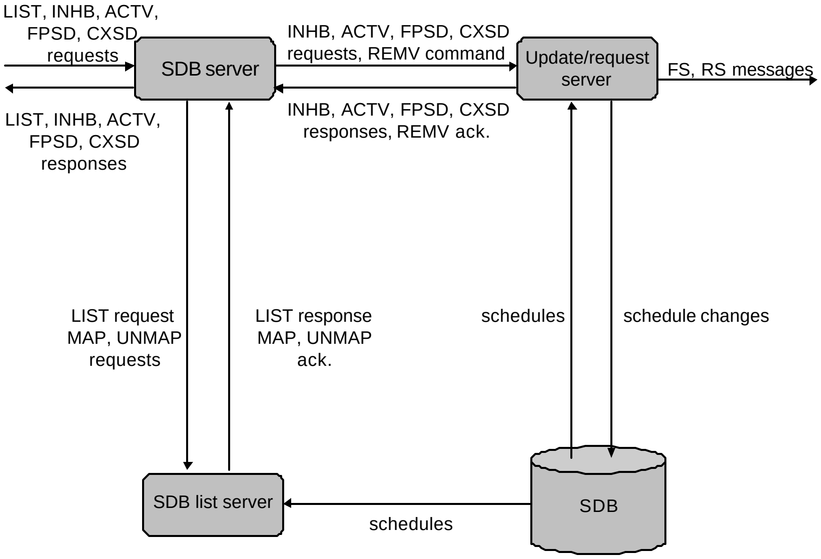

LIST, INHB, ACTV, FPSD, CXSD requests
SDB server
INHB, ACTV, FPSD, CXSD requests, REMV command
Update/request server
FS, RS messages
LIST, INHB, ACTV, FPSD, CXSD responses
INHB, ACTV, FPSD, CXSD responses, REMV ack.
LIST request MAP, UNMAP requests
LIST response MAP, UNMAP ack.
schedules
schedule changes
SDB list server
schedules
SDB

Figure 22-1. Overview of the SDB Processing Data Flow

In addition to responding to **INHB**, **ACTV**, Add Flight (**FPSD**), and Cancel Flight (**CXSD**) requests passed to it by the *SDB Server*, the *Update/Request Server* creates FS messages (modeled after FZ messages) from the flight schedules in the database. These newly created messages are sent in packets to the *NAS Distributor*, which then passes them on to the *Traffic Model*. The *Update/Request Server* sends FS messages independently of the *SDB Server*: If, for any reason, the communications link to the *SDB Server* fails, the *Update/Request Server* continues processing.

As well as passing messages along from external requests, the *SDB Server* passes **MAP** and **UNMAP** messages internally to the *SDB List Server*. The *Update/Request Server* gains write access to the database files to process **INHB**, **ACTV**, **FPSD**, or **CXSD** commands as described in the following steps:

(1) The *SDB Server* sends an **UNMAP** command to *the SDB List Server*, causing *SDB List Server* to release its access to the database files.

(2) The external request is sent to the *Update/Request Server*, where it is processed.

(3) Finally, the *SDB Server* sends a **MAP** command to the *SDB List Server* to regain access to the database files.

Another internal command is the **REMV** command which activates database housekeeping, changing any canceled flights back to active status after their period of cancellation has expired (24 hours). The *SDB Server* initiates this command when a timer event occurs at an interval determined by a run-time parameter to the process. It sends the **UNMAP** command to the *SDB List Server* and passes the **REMV** command to the *Update/Request Server* which runs the housekeeping routine. It sends the **MAP** command to the *SDB List Server* after it fields the reply from the *Update/Request Server*.

The *SDB List Server* receives all **LIST** and **CLIST** commands from the *SDB Server*. These are messages sent by the ETMS *listserver* in an internal format to provide data for **REQ LIST** requests made by the user through the *TSD* interface. The *SDB List Server* accumulates the requested flight data, stores it in a binary file, and notifies the *SDB Server* when it is done; the *SDB Server* then sends the file to the requesting *listserver*.

22.1 The SDB Server Process

Purpose

The *SDB Server* acts as a central source for receiving and sending messages, both within the *Schedule Database* function and in other parts of the system at the central and field sites. It also coordinates some of the routines needed to access and maintain the database.

Execution Control

The *SDB Server* is started along with the other programs that make up the *Scheduled Database* function (namely the *SDB List Server* and the *Update/Request Server)* by *nodescan*. The *SDB List Server* is invoked with one command line argument: the name of an ASCII file that contains all the necessary initialization parameters, which are described in the Input section.

The *SDB Server* process runs continuously; if it fails, it is restarted by *nodescan*. When non-fatal errors occur, an error message appears in the *sdb_server.pad.timestamp* window of the node on which it is running.

Input

The initialization parameters, read from the arguments file at program startup, are as follows:

- Directory in which the program will run.

- Name of the TCP/IP file to communicate with the *SDB List Server* and the *Update Request Server* and through which any cross-string communication will occur.
- *ETMS Communications* function logical process class name for the *listserver*.
- Time interval at which to perform database maintenance (in hours).
- Number of minutes to wait between checks that the connection with the *ETMS Communications* functions is still working.
- Name of the file with aliases for airports.
- Name of the file with site-specific authorization codes.
- Default site name to use for communicating through *ETMS Communications*.
- Number of cross-string *SDB* sites listed as next parameter(s).
- Optional argument for cross-string communication (the *ETMS Communications* site name for an *SDB Ser*ver on another string). There should be as many of these (separate lines) as indicated in the previous parameter.

Messages received from the *ETMS Communications* functions and from the internal *Schedule Database* function TCP/IP channels generate the main action of the program. The receipt of these messages is signaled by the triggering of system event counts. Other input to the *SDB Server* is also indicated by changes to the system event counts. This includes events triggered by the system clock and by messages being removed from the TCP/IP channels.

Output

Output from the *SDB Server* consists of the following:

- Messages sent through the *ETMS Communications* functions to the *listserver*.
- Messages sent through internal channels to the client processes.
- Completed file that contains the response to a **LIST** request.

Processing

During program startup, the *SDB Server* reads the parameters file described in the Input section. If the program cannot change to the directory named in the first input parameter, it writes a diagnostic message to the screen and proceeds, running in the current directory. If any other part of the initialization cannot be completed successfully, the program exits, writing an appropriate diagnostic message to the screen. Initialization is completed by opening all the necessary communications channels, initializing all the event counts that will be monitored, and loading all the necessary files and tables into memory.

After the initialization is complete, the *SDB Server* goes to sleep (waits for event counts to be triggered). When an event occurs, it is processed according to its type. The possible events are as follows:

- Internal *Schedule Database* function socket open channel request received - a message received from the client program. This is a request from the *SDB List Server* or the *Update/Request Server* to connect to the *SDB Server*.

- *ETMS Communications* functions message received from the *listserver* or *TSD* - messages handled according to their type, determined by the first word in the message text. **INHB**, **ACTV**, **FPSD**, and **CXSD** commands are sent through the appropriate channel to *Update/Request Server*. **LIST** requests are sent through the appropriate channel to *SDB List Server*.

- System timer event triggered - the first timer set up in initialization, occurs at whole-hour intervals, as indicated in the fourth input parameter. It initiates database maintenance operations. The database maintenance consists of reversing any cancellations (put in the database using the **CXSD** command) once their 24-hour time period has elapsed. *SDB Server* sends an internal command to *Update/Request Server* to initiate this action. Update/Request Server gains write access to the database files and calls the routine *remove_cancel* to update both the main database and the cancel index.

- System timer event triggered - the second timer set up in initialization, triggered every few minutes (the exact number of minutes is determined by the fifth input parameter). It causes a check to ensure that the connection with the *ETMS Communications* functions is still operating properly. If the connection has failed, reconnection is attempted. This timer event also triggers the sending of a **NOOP** message from *SDB Server* to the client programs (*SDB List Server* and *Update/Request Server*); they use the message to check the status of their socket connections to the *SDB Server*.

- Message removed from internal *Schedule Database* function socket - indicates that room may be available in the channel for messages queued up earlier when the channel was full. This event is important only when attempts to send messages to the client processes fail because the channels are full. When this happens, messages that cannot be sent are stored on a channel-specific queue. An event of this type signals the program that a message may have been removed from the channel, making room for more messages. The queued messages are then put into the channel in first-in first-out (FIFO) order until either the queue is empty or the channel is full again.

- Message received on internal *Schedule Database* function TCP/IP channels - a message sent from one of the client programs (a response to an **INHB**, **ACTV**, **FPSD**, **CXSD**, or **LIST** request). This event type occurs when one of the client processes puts a message in the internal channel. If the message is a response to an **INHB**, **ACTV**, **FPSD**, or **CXSD** command, that response is sent back to the *ETMS Communications* functions to be passed back to the original requestor. If it is an error message, *SDB Server* may change the message slightly before passing it back to make it more user-friendly.

- *Schedule Database* function message received from another *SDB Server* (cross-string) - a message sent from another *SDB Server* process running on another string. The messages they send are commands received on their own string that alter the contents of the database (**INHB**, **ACTV**, **FPSD**, and **CXSD** commands). This event type keeps databases across strings identical in case of the failure of one string and a switch-over to another.

Error Conditions and Handling

SDB Server writes all its diagnostic messages to the *sdb_server.pad.timestamp* window. Errors occurring during initialization are all fatal with the following exceptions: not being able to switch directories and any error that can be recovered using default values. An appropriate error message is written to the screen, and the program exits. The *sdb_server.pad.time stamp* is saved. Therefore, any messages written after the pad has successfully opened will be saved for examination. The error messages that can appear during initialization are the following:

- No filename passed in as argument.
- Cannot open arguments file.
- No TCP/IP file name in arguments file.
- No *listserver* class in arguments file.
- No time interval for db cleanup routines in args file.
- No time interval for polling frequency in arguments file.
- Bad alias filename.
- Cannot map alias file.
- No site authorization filename in args file.
- Cannot load site authorization codes.
- Cannot connect to node switch.

Errors in responding to requests may be due to some error in the request text, communications problems, or system problems. In each case, an error-specific message is written to the pad and also sent back to the requesting process. Many of the request-specific error messages are originally generated by *SDB List Server* or *Update/Request Server* and passed back by *SDB Server*. The specific errors passed back to the *ETMS Communications* functions to be returned to the appropriate *list server* are as follows for each command:

- **LIST**:
  - REQUEST error: Bad mode designator
  - REQUEST error: Unknown airport(s): *airport name*
  - REQUEST error: Bad date
  - REQUEST error: Bad start time
  - REQUEST error: Bad end time
  - REQUEST error: No name for LIST output file
  - REQUEST error: Cannot create/open LIST output file
  - REQUEST error: Cannot write to LIST output file
  - SYSTEM error: Error putting message in channel
- **INHB**:
  - INHB failed: Unknown flight id *flt_id*
  - INHB failed: Unknown flight id found in request ==> *text_string*
  - INHB failed: Unable to inhibit canceled flight
  - INHB failed: Bad date found in request ==> *text_string flt_id*
  - INHB failed: Unknown airline *airline identifier*
  - INHB failed: Unable to access schedule database
- **ACTV**:
  - ACTV failed: Unknown flight id *flt_id*
  - ACTV failed: Unknown flight id found in request ==> *text_string*
  - ACTV failed: Unable to activate canceled flight
  - ACTV failed: Bad date found in request ==> *text_string flt_id*
  - ACTV failed: Flight already active *flt_id*
  - ACTV failed: Unknown airline *airline identifier*
  - ACTV failed: Unable to access schedule database
- **CXSD**:
  - CXSD failed: Unknown flight id *flt_id*
  - CXSD failed: Unknown flight id found in request ==> *text_string*
  - CXSD failed: Flight is not effective within 12 hour range *flt_id*

  - CXSD failed: Bad time found in request ==> *text_string flt_id*
  - CXSD failed: Flight already canceled *flt_id*
  - CXSD failed: Unknown flight leg *flt_id*
  - CXSD failed: Unknown airport found in request ==> *text_string flt_id*
  - CXSD failed: Unable to access schedule database

- **FPSD**:
  - FPSD failed: Flight id already exists *flt_id*
  - FPSD failed: Bad date found in request ==> *text_string flt_id*
  - FPSD failed: It is past scheduled departure time for flight *flt_id*
  - FPSD failed: Unknown flight id found in request ==> *text_string*
  - FPSD failed: Unknown airport found in request ==> *text_string flt_id*
  - FPSD failed: Bad time found in request ==> *text_string flt_id*
  - FPSD failed: Bad ETE found in request ==> *text_string flt_id*
  - FPSD failed: Bad aircraft type found in request ==> *text_string flt_id*
  - FPSD failed: Leg with overlapping flight period already exists. Old leg: *dep_ap dep_time arr_ap arr_time*. New leg: *dep_ap dep_time arr_ap arr_time*
  - FPSD failed: Unable to access schedule database

If a **CXSD**, **FPSD**, **INHB**, or **ACTV** command is received from a site not authorized to issue commands that make changes to the database, the error message "This site not authorized to perform this command" is returned.

If an unrecognized message is received by the *SDB Server*, a message is sent back to the *ETMS Communications* functions echoing the first word of the command: "*word* failed: Unknown command."

22.2 The Update/Request Server Process

Purpose

The *Update/Request Server* has the following functions:

- It feeds FS messages to the *NAS Distribution* Process (*NAS.DIST*) to project flights a number of hours in advance before the actual departure time.

- It makes changes to the *SDB* itself at the request of traffic controllers. If a change affects any flights falling within a time period for which *Update/Request Server* has already generated an FS message, additional schedule messages or scheduled flight cancellation messages (RS) may be generated.

Execution Control

The *Update/Request Server* is started along with the *SDB Server* and the *SDB List Server* by *nodescan*. The *Update/Request Server* is invoked with one command line argument: the name of an ASCII file that contains all the necessary initialization parameters, which are described in the Input section.

The *Update/Request Server* runs continuously. If it fails and it was started as a child process, the *SDB Server* restarts it. On the other hand, if it was started by a script, *nodescan* restarts it.

Input

The initialization parameters, read from the arguments file at program startup, are as follows:

- Primary network addressing site.
- Secondary network addressing site.
- Name of the TCP/IP file to communicate with the *SDB Server*.
- Name of the file with pathnames of all the *SDB* database files.
- Interval at which to update *Monitor/Alert* (in minutes).
- Number of hours before flight departure to send data to *Monitor/Alert*.
- Number of hours to save FS or RS messages in a queue before deleting them.
- Directory in which the program will run.
- Name of file in which to save diagnostic messages relating to requests.
- Name of file in which to store update messages.
- Parameter that indicates whether to output update messages to a file.
- Parameter that controls the display on a processing pad.
- Name of file in which to store crash-related information.
- Parameter that controls whether to start generating FS messages from the time stored in the crash-related information file or from the current time.
- Maximum number of message buffers to be sent at any one time.

- Maximum number of update messages to be sent in the time interval equal to the number of minutes specified in the fifth input parameter.

Run-time input to the *Update/Request Server* is in the form of commands received from the *SDB Server*. These commands are as follows:

- **FPSD -** add new flight to *SDB* or edit existing flight.
- **ACTV -** activate flight.
- **INHB -** inhibit flight.
- **CXSD -** cancel flight.
- **REMV -** remove expired cancellations.

Update/Request Server also receives **NOOP** messages from the *SDB Server;* they are used for maintaining a connection between the servers.

Output

The main output of the *Update/Request Server* consists of large communications buffers filled with FS messages generated on a continuous basis. The program also generates RS messages in response to **INHB** and **CXSD** commands for flights whose FS messages were already sent; it generates FS messages for any flights added to the database with an **FPSD** command (which would have been sent out earlier if it had been in the database) and for previously inhibited flights activated by an **ACTV** command (flights for which FS messages should have been generated were the flights present and active within the last 24 hours).

- Processing Pad Output - contains all error messages, warnings, and FS and RS messages. Depending on the value of the twelfth input parameter, either all the FS messages generated will be shown or just the first message in every update buffer for every update.
- FS and RS Messages File - contains all FS and RS messages generated, *if created*. (The creation of this file depends on the value of the eleventh input parameter.)
- Requests file - contains the requests that failed and appropriate diagnostic messages.
- Crash-related information file - contains the ASCII departure time for the last flight with a successfully transmitted FS.

FS Message Format

An FS consists of character data items separated by spaces. The first data item in a message is the encoded time of message creation. It is expressed in ASCII-coded seconds and therefore is likely to contain unprintable characters. The rest of the items in the FS message are in readable ASCII characters. The data items are described in the following list in the order in which they appear in the message. (Lengths are given as maximum possible values; actual lengths could be shorter.)

(1) 4-character timestamp followed by string **XFS**.

(2) 7-character flight ID followed by /, followed by 3-character computer ID. The computer ID consists of a letter from **A** to **Z** followed by a number from **01** to **99**, and it is generated sequentially for every FS message. The number part is incremented before the letter part.

(3) 4-character aircraft type.

(4) 4-character cruising airspeed in nautical miles/min * 100.

(5) 4-character departure airport.

(6) **P** followed by a 4-character departure time. (**P** stands for planned.) The first two digits of the departure time represent hours; the last two, minutes.

(7) 3-character cruising altitude in hundreds of feet.

(8) 512-character field10 followed by /, followed by 4-character estimated time of arrival (ETA).

(9) 4-character Julian departure date.

RS Message Format

An RS consists of character data items separated by spaces. The first data item in a message is the encoded time of message creation. It is expressed in ASCII-coded seconds and therefore is likely to contain unprintable characters. The rest of the items in the RS message are in readable ASCII characters. The data items are described in the following list in the order in which they appear in the message. (Lengths are given as maximum possible values; actual lengths could be shorter.)

(1) 4-character timestamp followed by string **XRS**.

(2) 7-character flight ID followed by /, followed by 3-character computer ID. Computer ID consists of a letter from **A** to **Z** followed by a number from **01** to **99**, and it is generated sequentially for every RS message. The number part is incremented before the letter part.

(3) 4-character departure airport.

(4) 4-character arrival airport.

(5) 4-character Julian departure date.

(6) 4-character departure time. The first two digits of the departure time represent hours; the last two, minutes.

Processing

When *Update/Request Server* starts executing, it processes the command line parameters file, and it maps the *SDB* files (see Figure 22-2). It then opens connections to the *ETMS Communications* functions (the first or second input parameters) and the *SDB Server* (the third input parameter) and initializes their event counts. If the program encounters any errors during these steps, it exits. Otherwise, it enters an infinite wait loop where it performs dispatches when corresponding event counts are triggered.

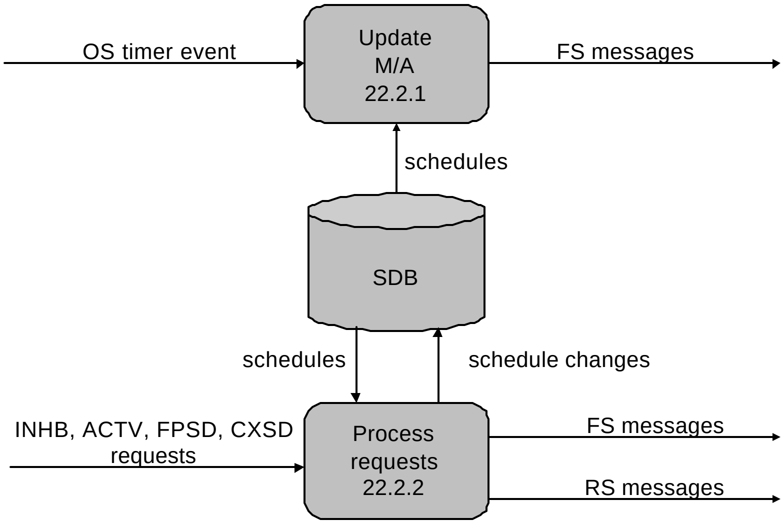


Figure 22-2. Data Flow of the Update/Request Server Process

Under *normal* circumstances, **resend_mode** is **False**. It is set to **True** when a message to *NAS.DIST* is returned because *ETMS Communications* and/or *NAS.DIST* are down. The strategy is to keep trying to reconnect to the *NAS.DIST*, and to keep sending a dummy RS message to the *NAS.DIST* until it is not returned. At that point **resend_mode** is set to **False**, the originally returned message is resent, and the process of generating and sending messages to the *NAS.DIST*, as well as advancing the crash-related information file time (the last time at which messages were generated) is resumed, without having to keep track of the generation time, of which messages were sent and returned and must be resent, etc.

The different events that can occur are as follows:

- The ***Request_Get_Ec*** or ***Msg_Get_Ec*** event count **-** triggered by a request from the SDB Server. This event count causes the *Update/Request Server* to dispatch the *Process Requests* module.
- The ***T15_Ec*** event count **-** invokes the *Update Monitor/Alert* module. Its frequency is determined by the fifth input parameter (which used to be hard-wired at 15 minutes).
- The ***T1_Ec*** event count **-** invokes every minute either the *flush_updates* routine if **resend_mode** is **False**, or the *sdb_check_resend* routine if True. Both routines try to verify the connection to the *NAS.DIST*, and both attempt to reconnect, if necessary. *Sdb_check_resend* may toggle **resend_mode**. If the connection to *NAS.DIST* is intact (or has been re-established) and there are any messages waiting in the *Updates* queue, both routines attempt to transmit them.
- The ***T2_Ec*** event count **-** invokes the *upkeep* routine every two minutes. It consists of several checks to ensure that all necessary connections are in order. If *Update/Request Server* does not have write access to the *SDB*, it tries to map the *SDB* for writing. If there is no connection to the *SDB Server*, it tries to reestablish it. Then, if there are any requests waiting in the *Request* queue, it dispatches the *Process Requests* module.
- The ***Timeout_Req_Ec*** **-** maintains connection to the *SDB Server*. If the *Update/Request Server* does not receive a **NOOP** message from the *SDB Server* within a time period related to an *ETMS Communications* constant, this event count triggers a reconnection to the *SDB Server*. The old connection is closed and an attempt to reconnect is then made.

22.2.1 The Update Monitor/Alert Module

Purpose

The *Update Monitor/Alert* module makes flight schedule information available to the central site *User Interface* function within a number of hours before the flight departs. The number of hours is determined by the third input parameter in this module's Input section. See Figure 22-3 for the logic flow of the *Update Monitor/Alert* module.

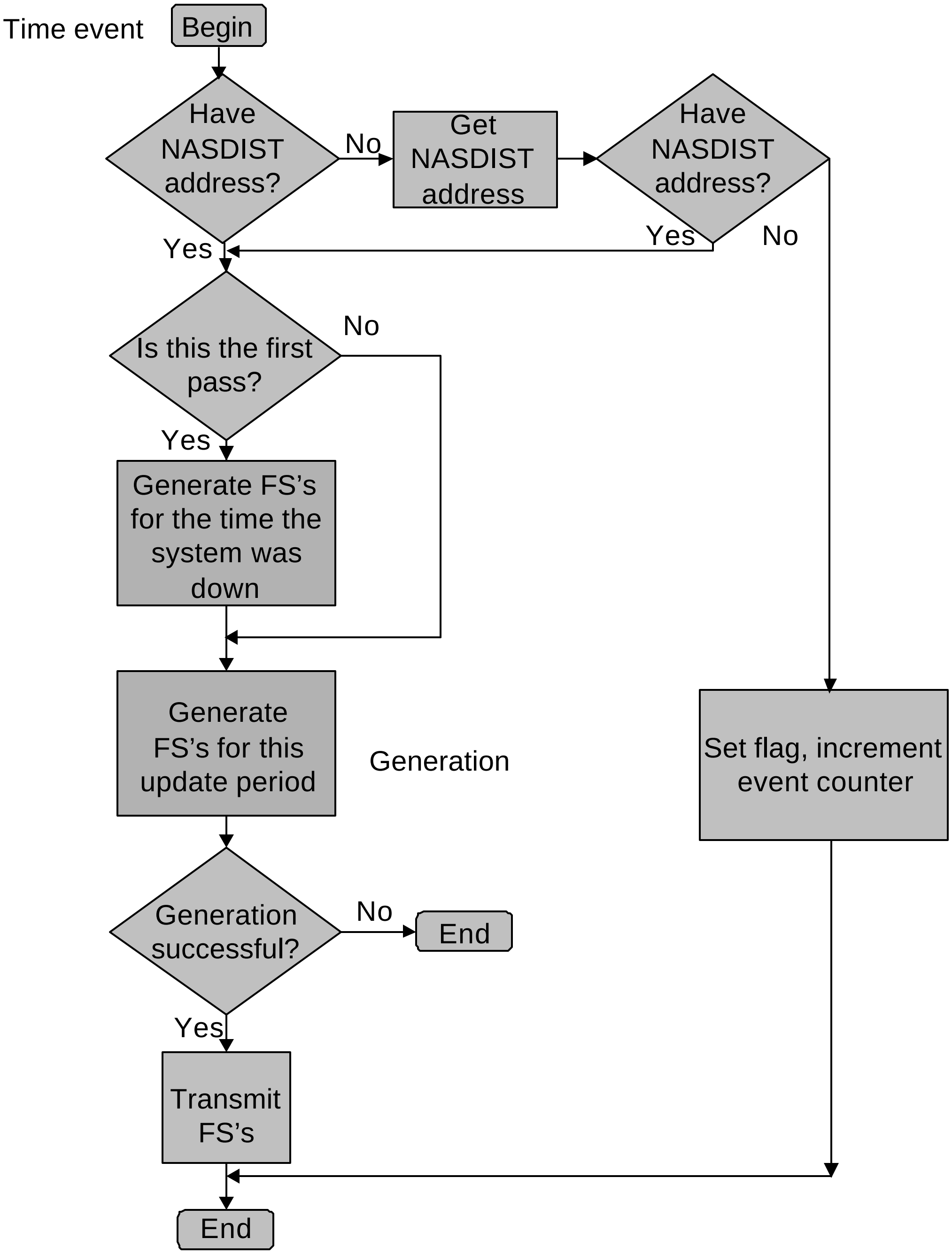


Figure 22-3. Logic for the Update Monitor/Alert Module

Input

The *Update Monitor/Alert* module reads flight data from the *SDB*. It also accepts the following parameters which the *Update/Request Server* reads from its argument file:

- Function site name to connect to *ETMS Communications*.
- Interval at which to update *Monitor/Alert* (in minutes).
- Number of hours before flight departure to send data to *Monitor/Alert*.
- Number of hours to save FS or RS messages in a queue before deleting them.
- Name of file in which to store update messages.
- Parameter that indicates whether to send update messages to a file.
- Parameter that controls the display on a processing pad.
- Name of file in which to store crash-related information.
- Parameter that controls whether to start generating FS messages from the time stored in the crash-related information file or from the current time.
- Maximum number of message buffers to send at any one time.
- Maximum number of update messages to transmit in a single period of time determined by the interval parameter.

Output

The main output of the *Update/Monitor Alert* module consists of large communications buffers filled with FS messages generated on a continuous basis.

The *Update/Monitor Alert* module optionally sends FS messages to the process pad and the FS messages file. It also sends to the process pad the total number of FS messages generated and the number of them successfully transmitted in each batch. Error messages for errors that occur while this branch is active also go to the process pad. After the *Update Monitor/Alert* module transmits a buffer of FS messages, it writes the time stamp of the first FS message in the buffer to the crash-related information file.

Processing

Preprocessing - The time event count for the *Update Monitor/Alert* module is incremented so its next invocation always coincides with the beginning of the next time slot. (In other words, the invocation of *Update Monitor/Alert* is independent of the length of time it takes to process one time slot.) Thus, if the time event is set to occur every 15 minutes, *Update/Monitor Alert* is always invoked at the 0th, 15th, 30th and 45th minute of the hour.

However, if **resend_mode** is True (because *ETMS Communications* and/or *NAS.DIST* are down) *Update Monitor/Alert* does not generate or send any messages, nor check the connection to *NAS.DIST* **-** instead, it sets the ***T15_Ec*** event count to occur sooner (just as if it were trying to send multiple buffers) and returns control to the *Update/Request Server.*

When *Update Monitor/Alert* is invoked, it first checks to see if it has *NAS.DIST*'s address. If it does, it proceeds further. Otherwise, it tries to get the address. If it fails, it sets a flag, increments the normal update period event count, and returns control to the *Update/Request Server.*

On the first invocation, if the ninth input parameter was set to start generating messages based on the time in the crash-related information file, *Update Monitor/Alert* generates FS messages that should have been generated while the system was down. This is done by using the time stored in the crash-related information file as the starting point from which to generate messages. After each normal transmission, the system stores (in the crash-related information file) the departure time of the flight for which an FS message was just sent. Thus, the system can always regenerate FS messages lost from the *Updates* queue when the system went down.

Prior to generating new FS messages, the system reports the number of unsent messages (if any) stored in *Updates* queue. These are FS messages that could not be transmitted before, either because there was no connection to *NAS.DIST* or because the program generated more FS messages than the maximum throughput of the system.

Routines - The *Update Monitor/Alert* module has two routines: *build_update_message* and transmit:

- *build_update_message* routine **-** After the preliminary steps described in the Preprocessing section have been taken, the system generates and enqueues FS messages for this update. When this routine is invoked the first time, the system calculates a pointer to the first applicable flight record in the *SDB*. If the corresponding parameter was specified and *build_update_message* was called to regenerate the FS messages missed while the system was down, the advance time is added to the time from the crash-related information file to calculate the starting update time. Otherwise, the advance time is added to the current time. The call to *get_time_table* returns the pointer to the first flight record in the corresponding time slot.

  The ending time for the update is always calculated by adding the update period length to the current time plus the advance time. After this, the system proceeds to generate and add FS messages to the update buffer. If the buffer gets full, the system adds it to the *Updates* queue and creates a new buffer. This is done by a dedicated set of routines called from *build_update_message.* When the departure time in *SDB* record matches this update's ending time, FS generation is ended. A pointer to this *SDB* record is saved. The next time the *build_update_message* routine is called, the FS generation continues from this record. Thus, if the *build_update_message* routine is not called for several update periods (because of some problem), the FS generation will pick up where it ended the last time. When the time reaches 2400, the pointer in the *SDB* is reset to the beginning of the *SDB.*

If there is no connection between the *Update/Request Server* and the *NAS.DIST* for a sufficiently long period of time, the FS messages accumulating in the *Updates* queue become outdated and are discarded. This time period (in hours) is set in the fourth input parameter.

- *transmit* routine - attempts to transmit the update buffers stored in the *Updates* queue to the *NAS.DIST* one at a time. If transmission is successful, the FS messages from that buffer are output to the processing pad and to the FS messages file. The time stamp of the first message in the buffer is copied into the crash-related information file.

  If after transmission, there are still FS messages waiting in the *Updates* queue, the one-minute event count is incremented to attempt transmission one minute later (see Figure 22-4).

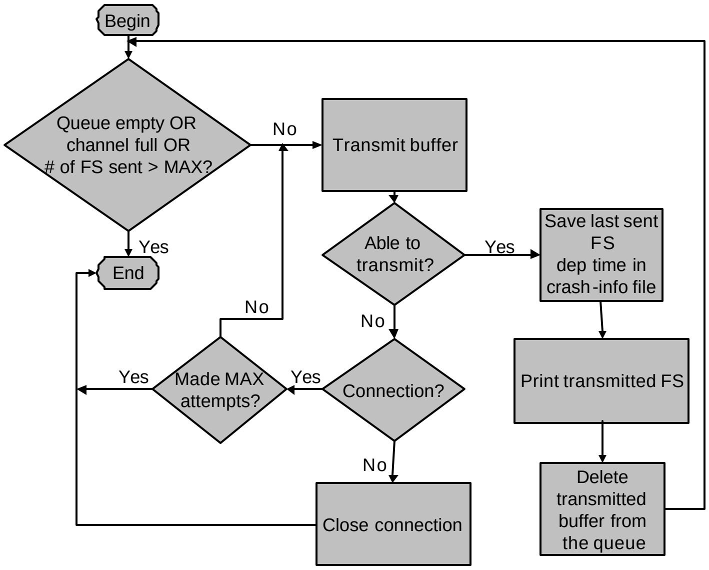

Begin
Queue empty OR channel full OR # of FS sent > MAX?
No
Transmit buffer
Yes
End
Able to transmit?
Yes
Save last sent FS dep time in crash-info file
No
No
Made MAX attempts?
Yes
Yes
Connection?
Print transmitted FS
No
Close connection
Delete transmitted buffer from the queue

Figure 22-4. Logic for the Transmit Routine

Error Conditions and Handling

(1) If there is no connection to the node switch, the process tries to reconnect repeatedly (once a minute). If *NAS.DIST* can't be found, the process keeps trying to get its address.

(2) If some other error occurs during transmission, two more attempts to transmit are made before exiting.

22.2.2 The Process Requests Module

Purpose

The *Process Requests* module processes requests from the *TSD* to make changes to the *Schedule Database.* As a result of these changes, corrective FS or RS messages can be generated and sent to the *Monitor/Alert* function (see Figure 22-5).

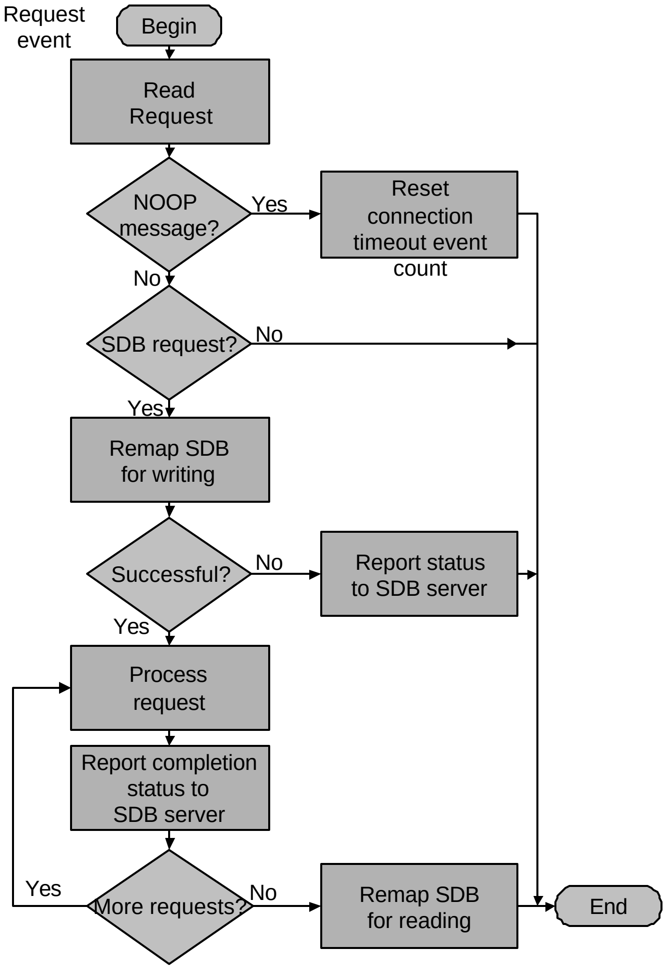


Figure 22-5. Logic for the Process Request Module

Input

The *Process Requests* module accepts the following parameters that the *Update/Request Server* reads from its argument file:

- Name of the TCP/IP file to communicate with the *SDB Ser*ver.
- Name of the file in which to save diagnostic messages relating to requests.

The *Process Requests* module receives the following messages from the *SDB Server*:

- **INHB -** inhibit a flight.
- **ACTV -** activate a flight.
- **CXSD -** cancel a flight.
- **FPSD -** add a new flight to the *SDB* or edit an existing flight.
- **REMV** - reverse an expired cancellation.

Output

- When a request is successfully executed, the *Process Requests* module sends a success message to the *SDB Server*. Otherwise, it sends a failure message.
- When a request fails, the request message, the time it was received, and the error message are saved in the requests diagnostics file (the second input parameter).
- One or more FS/RS messages may be generated depending on a request.

Processing

The *Process Requests* module handles four different request commands covered in the following order in the indicated sections:

- **INHB -** processed by the *process_inhibit* routine **-** Section 22.2.2.1
- **ACTV -** processed by the *process_activate* routine**-** Section 22.2.2.1
- **CXSD -** processed by the *cancel_flight* routine **-** Section 22.2.2.2
- **FPSD -** processed by the *add_flight* routine **-** Section 22.2.2.3

Another command accepted by this module, the **REMV** command, is an internal command sent by the *SDB Server* process to initiate a database maintenance routine.

Routines **-** The *Process Request* module uses the following routines which apply to all the module commands:

- *sdb_get_indices* routines

- *get_air_carrier* routine has for input the air carrier name and the time of day of the desired time bucket. The routine uses *get_hash_air_carrier_ptr* to locate the air carrier in the **hash map file** and obtain the offset to the linked list of flights for the air carrier; the offset is then returned to the calling routine.
- *get_arr_airport* routine has for input the arrival airport name and the time of day of the desired time bucket. The routine uses *get_hash_arr_airport_ptr* to locate the arrival airport in the **hash map file** and obtain the offset to the timetable for the arrival airport. The offset to the correct time bucket is calculated using the time-of-day input. The offset to the linked list of flights for that time bucket is then retrieved and returned to the calling routine.
- *get_dep_airport* routine has for input the departure airport name and the time of day of the desired time bucket. The routine calls *get_hash_dep_airport_ptr* to locate the departure airport in the **hash map file** and obtain the offset to the time table for the departure airport. The offset to the correct time bucket is calculated using the time of day input. The offset to the linked list of flights for that time bucket is then retrieved and returned to the calling routine.
- *get_flight_id* routine has for input the flight ID. The routine uses get_hash *flight_id_ptr* to locate the flight in the **hash map file** and retrieve the offset to the linked list of legs for that flight; the offset then is returned to the calling routine.
- *get_time_table* routine has for input the time as a four-digit integer. The routine then locates the time table slot for that time and returns the offset to the flight in the *SDB* from that time slot.

- *sdb_indices_util* routines
  - *create_map_file* routine creates an empty map file and initializes all the fields in the entire map file. The type of the map file is defined by the **map_type** variable. The field values are similarly defined by the type of the map file.
  - *get_hash_dep_airport_ptr* routine returns the pointer in the departure airport **hash map file** to the specified departure airport or to the next empty bucket.
  - *get_hash_flight_id_ptr* routine returns the pointer in the flight id **hash map file** to the specified flight id or to the next empty bucket.
  - *open_map_file* routine opens an existing map file for exclusive write so it can be updated.
- *sdb_add action* routine has for input the type of action (add, cancel, or inhibit) and the *SDB* offset of the flight to be added. The routine first opens the action

list map file appropriate for the action type. If the file does not exist, it is created. The flight is added to the linked list in alphabetical order, and the map file is closed.

- *sdb_remove_action* routine has for input the type of action (add, cancel, or inhibit) and the flight ID of the flight to be removed. The routine first opens the action list map file appropriate for the action type. The flight is then located on the linked list and removed. If the action is a cancel, the **cancel bit** is set to **zero** in the *SDB* flight record, and the map file is closed.

22.2.2.1 INHB/ACTV Command Processing

When the **INHB** or **ACTV** requests are received, the entire message - consisting of the command word (**INHB**, **ACTV**), the flight or airline ID, and the optional start and end dates - is broken down into its components by the *inhibit_activate_delete* routine. The program computes the start and end dates in the same way as for the **FPSD** requests (see Section 22.2.2.3). If the request contains a flight ID, the program calls the *get_ flight_id* routine, which returns a pointer to the list of legs for the flight ID in question. If, however, an airline name is specified, the program makes a call to the *get_air_carrier* routine which returns a complete list of flights for that airline. The list is then processed by either *process_inhibit* or *process_ activate* depending on the request type. No operation is permitted on a canceled flight, and such a request would produce an error message. Otherwise, a flight is inhibited or activated as described in the next section. An RS message is generated for **INHB** if, by the time it is received, an FS message has been already sent for the flight for that date. As a result of a request, a FS message may be generated for the flight leg in case an **ACTV** is due by that time. An *inhibit* bit is set or cleared in the flight record depending on the request type and whether or not the flight is still inhibited for some interval. A message containing a list of inhibit periods before and after the request was processed is returned to *SDB Server.*

Inhibit List Processing - In addition to a list of periods in which a flight is defined to operate (the effective/discontinue dates), each flight has a second independent inhibit list. Even if the flight is defined to fly at a specific date on the first list, it is not operational if this date appears on the inhibit list. Such dual structure provides the capability to temporarily inhibit and activate a flight without affecting its predefined original operational period. An inhibit list is implemented as a list of nodes, each of which contains starting and ending dates for the period represented by the node. These intervals are noncontiguous and discrete; this means there is at least one day on which a flight would be operational (if it is operational according to the first list) between any two intervals on the inhibit list. When the *SDB* is created from the raw input data there are no inhibit lists associated with the flight records. As the live *SDB* is subjected to **INHB** and **ACTV** requests, the inhibit list gets created, grows, shrinks, disappears, and gets created anew. All manipulation of the lists is handled by *substruct_period_from_ed_list* and *add_period_to_ed_list* routines. The first routine computes set exclusion and the second computes set union between the list and the period in request. As a result of the operation:

- a new node may be added to the list
- one or several nodes may be deleted, and/or
- an existing node might have its dates reset

If a node on the list starts to overlap in time with other list nodes, the whole list is adjusted to reassert noncontiguousness and discreteness. The flight is considered to be active only when there are no periods on its inhibit list (when its inhibit offset is *null*). Only then is the **inhibit status bit** cleared.

22.2.2.2 CXSD Command Processing

The *cancel_flight* routine processes the **CXSD** request. The **CXSD** request is valid for flights departing within a 24-hour range from the time the request is received, plus or minus 12 hours from receipt of the request. Hence, the possible departure date can be within a two-day interval. The precise date is determined in the following fashion:

(1) Twelve hours is subtracted from the current time to obtain the lower end point for the range - this value is in the system's internal time format and thus contains the date as well as the time.

(2) This time is then compared to the departure time in the flight record.

(3) If the time in the flight record is equal to or greater than the lower end point time, the flight is operative on the lower end point date.

(4) If not, the flight is operative on the upper end point date, which is computed by adding 12 hours to the current time.

When the flight is canceled, its **status bit** is set and remains so for 24 hours. During this time, the flight is invisible to the outside world: the record cannot be affected by any *SDB* commands, and the flight data does not appear on any reports or be sent scheduled updates.

After 24 hours, the **status bit** is cleared and the flight regains its normal status; this is done by *remove_cancel* routine, which is triggered internally by the software at regular intervals. The *remove_cancel* routine first opens the cancel (action) list map file. All the flights are then checked on the linked list of canceled flights to determine which have been canceled for 24 hours or more. These flights are removed from the linked list, and the **cancel bit** is set to **zero** for each flight in the *SDB* flight record. The map file is then closed.

Error Conditions and Handling

- If the connection to the *SDB Server* is lost, the *Process Requests* module exits. The connection is re-established by the *upkeep* routine, invoked every two minutes.

- If the *Process Requests* module cannot remap the *SDB* with the read-with-intent-to-write (RIW) lock, it exits. Another attempt to change the lock is made by the *upkeep* routine, invoked every two minutes.
- If a request cannot be carried out on the flight because of the flight status, a failure message is returned. For example, no request can be carried out on a canceled message, except **REMV**.
- If a received request contains syntax errors or an error occurs in responding to a request, the request message is saved in the requests diagnostics file with the time it was received. A failure message is returned to the *SDB Server*.

22.2.2.3 FPSD Command Processing

Purpose

The **FPSD** command is issued to add a new flight, add a new flight leg for an existing flight, or to edit an existing flight.

FPSD Command Format

The **FPSD** command has the following format:

flt_id dep_ap dep_time arr_ap ete acft_code days start_date end_date

The **days**, **start_date**, and **end_date** are optional. For any optional argument to be parsed, the optional arguments preceding it must all be present. Any missing arguments are computed as described in the following four cases:

NOTE: If a request, when received, *is past departure time* for that day and the start_date is today, it is promoted to tomorrow. If the days of operation are for *one day only*, tomorrow's day of the week is set. Bear this in mind when reading the following descriptions.

(1) No optional arguments - **days** get set to the day of the week on which the request is received. (For example, if the **FPSD** is received on Monday, **days** are set to **OXOOOOO**.) Both the **start date** and the **end date** are set to the date on which the **FPSD** was received. The flight is added for one day.

(2) **days** are present - has the same effect as the first case.

(3) **days** and **start date** are present - weekdays of operation are set to **days**. The flight is set to be effective from the **start date** until one year after the date on which the current *SDB* was created (known as the *maximum discontinue date*).

(4) **days**, **start date**, and **end date** are present - flight is set to operate on **days** of the week from the **start date** to the **end date**.

Top Level FPSD Command Processing -The top level processing of the **FPSD** command begins with the *add_flight* routine (see Figure 22-6). Here the buffer received from the *SDB Server* is parsed into its constituents. The routine computes the primary departure and the arrival airport names to be stored in the database, as well as defaults for the three last optional arguments for the **FPSD** command. Then other necessary information, such as an index into the aircraft type file and an offset into the aircraft category file, is computed within the *process_add* routine.

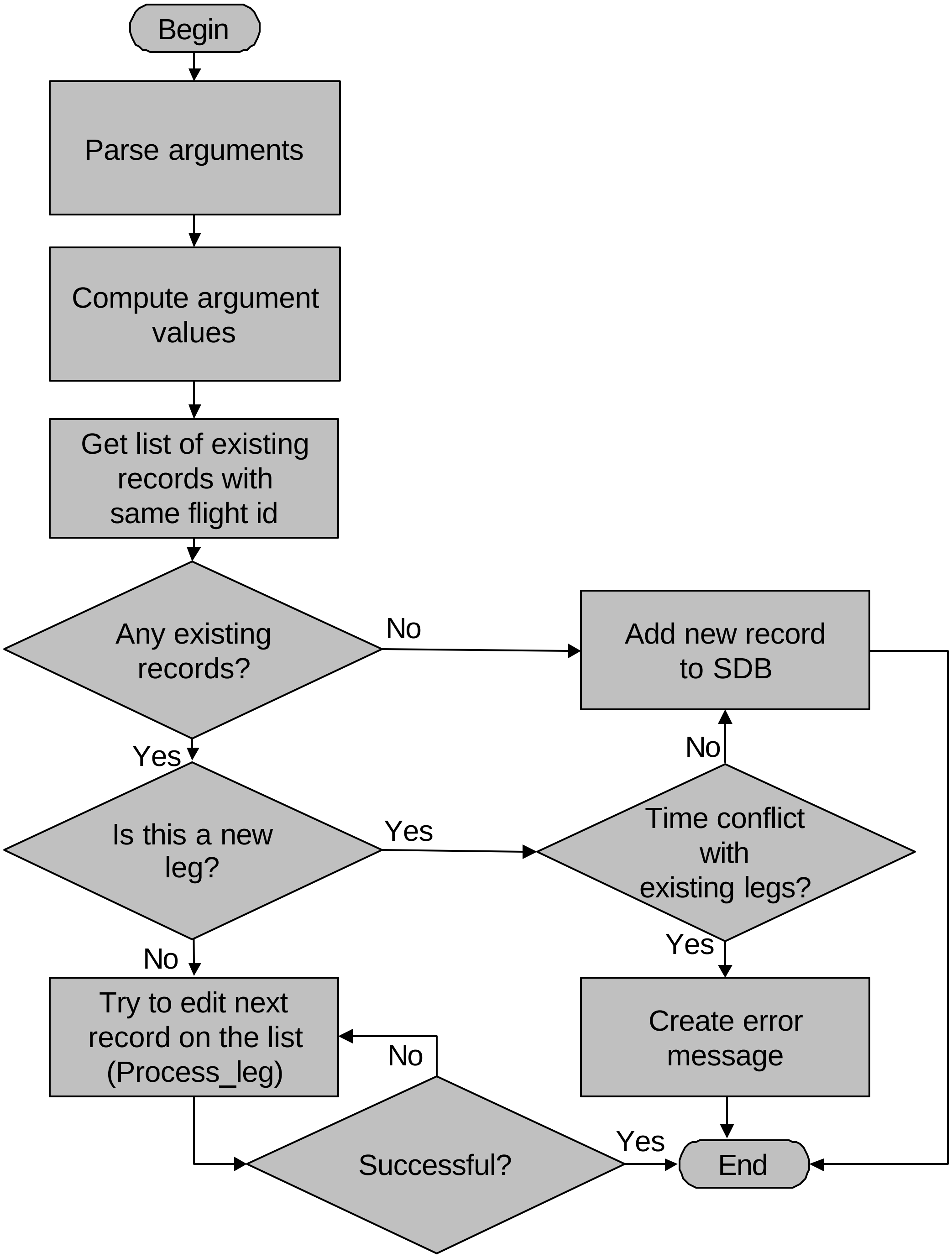


Figure 22-6. Logic for the add_flight Routine

In the same routine, a decision is made on whether the request implies the addition of a new record or the editing of an existing one. The *get_same_flight_id_list* routine returns two pointers to two lists of flight legs with the same flight ID as in the **FPSD** request: the first list contains legs with flights for the same departure and arrival airport pair; the second list contains different legs. If the first list is *not* empty, then the *process_leg* routine is used on the entire list to edit an existing leg. If the first list *is empty*, then the *dep_arr_times_conflict* routine is used to check if a time en route period in the **FPSD** overlaps with any of the legs on the second list. If such an overlap is detected, the **FPSD** returns a relevant error message since the flight cannot be defined on more than one leg during the same time interval. If there is an overlap, the *add_new_record_to_sdb* routine is called to add a new flight leg.

Adding a New Flight/Flight Leg - The *add_new_record_to_sdb* routine adds a new flight record at the end of the *SDB* table mapped file. The mapped file address is obtained by looking at the **event_route_off** field in the first record of the file containing its current length in bytes. All computed and derived flight data is then assigned to that location by a call to the *fill_sdb_record* routine. Next, a call to the *search_sdb_to_add* should return pointers to the records preceding and following the one to be added, based on the departure time and the flight ID order.

If *search_sdb_to_add* finds an existing record with the same data as in the **FPSD** command during the search, it returns **False**. This indicates an inconsistency between the flight ID index file and the main *SDB* file. In this case, the program terminates with the *SDB integrity violated* message. If the *search_sdb_to_add* routine returns **True**, the *sdb_add_indices* routine is called to update the index files. If they cannot be updated, the program terminates with an *SDB integrity violated* message.

For information on the logic flow for the *add_new_record_to_sdb* routine, see Figure 22-7.

The *sdb_add_indices* routine calls the following routines to add the flight to each index:

- The *add_time_table_index* routine has for input the offset in the *SDB* of the new flight being added. The routine compares the departure time of the new flight to the flight in the corresponding time bucket: (1) if the new flight has an earlier time than that of the flight in the time bucket, the offset to the new flight replaces the offset to the old in the time table index; (2) if both flights have the same time, the flight IDs are compared and if the new flight has a flight ID alphabetically before the old, the offset to the new flight replaces the offset to the old in the time table index.

- The *add_flight_id_index* routine has for input the offset in the SDB of the new flight being added. The routine first determines if the flight is in the **hash map file**: (1) if it is not, the flight is added to the **hash map file** and it is put on a linked list by itself; (2) if it is already in the **hash map file**, a new leg is added in the correct position on the linked list for this flight, according to departure time.

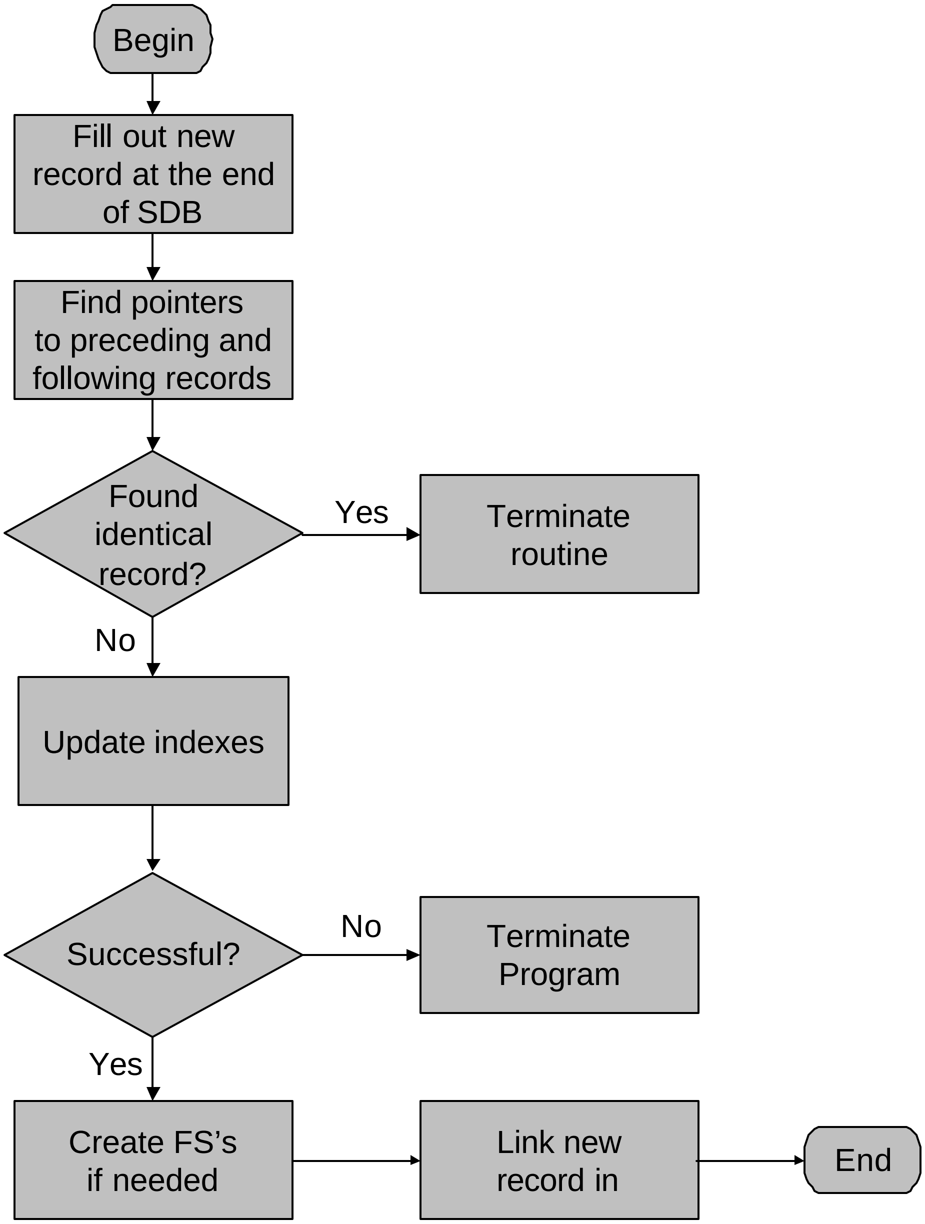


Figure 22-7. Logic for the add_new_record_to_sdb Routine

- The *add_air_carrier_index* routine has for input the offset in the *SDB* of the new flight being added. The routine first determines if the air carrier of the new flight is already in the **hash map file**: (1) if it is not, the air carrier is added to the **hash map file** and the flight is put on a linked list by itself; (2) if the air carrier is already in the **hash map file**, the time of this flight is compared to that of the flight in the linked list; if time and flight ID are the same, it returns with a "duplicate flight" error code; otherwise, the new flight is added to the linked list for this air carrier, according to departure time.

- The *add_arr_airport_index* routine has for input the offset in the *SDB* of the new flight being added. The routine first determines if the arrival airport of the flight is in the **hash map file**: (1) if it is not, the arrival airport and the offset in the timetable for the new flight are added to the **hash map file** and the new flight is put on the arrival airport linked list by itself; (2) if the arrival airport is already in the **hash map file**, the time of this flight is compared to that of the flight in the linked list; if time and flight ID are the same, the flight is not added; otherwise, the new flight is added to the linked list for this arrival airport, according to arrival time.
- The *add_dep_airport_index* routine has for input the offset in the *SDB* of the new flight being added. The routine first determines if the departure airport of the flight is in the **hash map file**: (1) if it is not, the departure airport and the offset in the time table of the new flight are added to the **hash map file** and the new flight is put on the departure airport linked list by itself; (2) if the departure airport is already in the **hash map file**, the time of this flight is compared to that of the flight in the linked list; if time and flight ID are the same, the flight is not added; otherwise, the new flight is added to the linked list for this departure airport, according to departure time.

If processing continues successfully, an FS message may be generated for a newly added flight. The new record is linked into the *SDB* by storing its offset in the preceding record and setting its *offset_to_next* field. This process is concluded by writing out the length of the *SDB* mapped file into the first record.

Editing an Existing Record - An *SDB* database may contain more than one record for the same flight leg. It is possible, for example, for a flight to fly from the first to the ninth of the month on Tuesdays and Thursdays; from the tenth to the thirtieth on Mondays, Wednesdays, and Fridays; and from the twenty-first to the thirtieth on Saturday and Sunday. The flight data is stored in the *SDB* in the form of three separate records, identical except for differently set bits for the weekdays of operation in the **status_bits** field and offsets into different lists of effective periods of operation.

As long as two records have the same departure and arrival airports, they represent the same flight leg. Besides days of the week, the other ways in which flight data could vary while the flight record still represents the same leg are equipment type and departure and/or arrival time. These four data categories (described on the next page) can be edited by the **FPSD** command (see Figure 22-8). Only one change is allowed per request, except for departure and arrival times, which can be changed simultaneously, and a special case (described in this subsection). If more than one change is required, enter the necessary **FPSD** commands successively to implement them. (This one-change requirement permits the program to determine which of several possible *SDB* records needs editing.) The program steps through the list of flight legs until it finds one whose flight data differs from the data in the **FPSD** in only one allowed category. It edits that record, and when editing is completed, the flight leg list processing terminates and the *Update/Request Server* returns a status reply message.

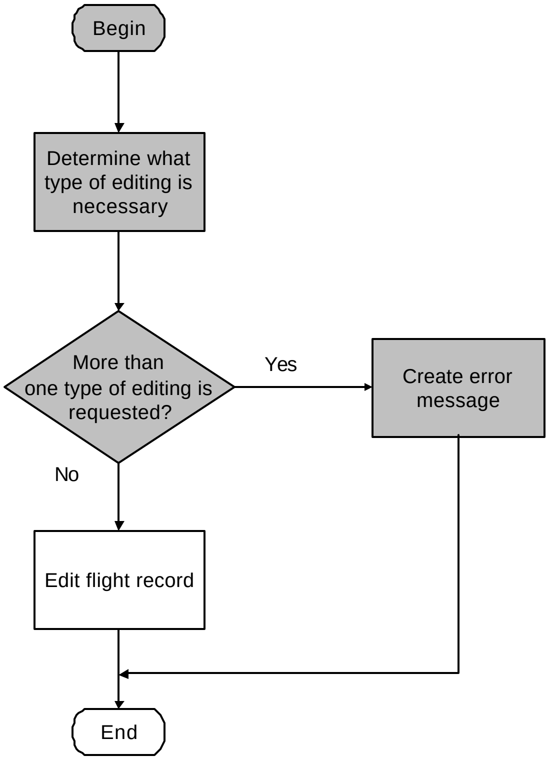

Begin
Determine what type of editing is necessary
More than one type of editing is requested?
Yes
Create error message
No
Edit flight record
End

Figure 22-8. Logic for Editing an Existing Record

The following list shows how the changes allowed by the **FPSD** command are implemented:

- *Equipment* - Change an equipment index in the *SDB* record to point to the entry in the equipment type file containing the new aircraft type.

- *Weekdays of Operation* - Reset the bits in the **status_bits** to the new days of operation.

- *Dates of Operation* - Reset the offset in the effective dates list to point to a list containing a single interval derived from the dates in the **FPSD** command, regardless of the contents of the old list.

- *Departure and/or Arrival Times* - If the request is to change the arrival time alone, it is done by resetting that field in the *SDB* record. If the request is to change the departure time, that field in the *SDB* is changed. However, since the *SDB* is sorted on departure time, simply editing the arrival time field in the record would destroy the order. The actual steps taken are as follows (see Figure 22-9):

(1) The *Search_Sdb_To_Add* routine is called to return pointers to records preceding and following the one being edited.

(2) The two records are linked around it.

(3) Since the change involves departure time, an RS message may be generated, if appropriate, and the indexes for the record are purged from the index files; if this can't be done, the program terminates abnormally with an *SDB integrity violated* message; otherwise, the departure and arrival times get reset.

(4) The *search_sdb_to_add* routine is called to return pointers to the records preceding and following the one being edited, based on the new value of the departure time.

(5) The record is linked in by resetting its offsets and the offsets of the preceding record (and FS message may be generated at this point).

(6) The *sdb_add_indices* is called to reset indexes; if it fails, the program terminates abnormally with an *SDB integrity violated* message.

- *Special Case* **-** A request to change departure time may cause the effective date to be shifted one day forward or backward, resulting in a requested double change. Consider a flight departing at 10 A.M. on June 11, added for a period of more than one day at 1 P.M. on June 11. Since it was already past the departure time when the **FPSD** command was received, the effective date is stored as June 12, even if June 11 was specified in the request. If at 3 P.M. on the same day (June 11) somebody tries to change the departure time to any time past 3 P.M. (5 P.M., for example), the system accepts June 11 as the effective date without attempting to promote it. However, it then becomes a request to change both the departure time (from 10 A.M. to 5 P.M.) and the effective date (from June 12 to June 11). Thus, a presumably *valid* request would be rejected because only *one* change is allowed per request.

  The same situation occurs if a flight is added initially with an effective date for the same day. Any attempt to edit the departure time when it is actually past departure time moves the effective date one day ahead.

  To avoid this conflict, the system uses the following stratagem. When *qualify_editing* routine (called to check the changes requested) detects a requested change to both the departure time and the effective date, the *dep_time_effect* routine is called to check if a difference between the effective dates actually exists. If it does *not* or if the flight has more than one date pair on the effective period list, the request is rejected; otherwise, it is allowed.

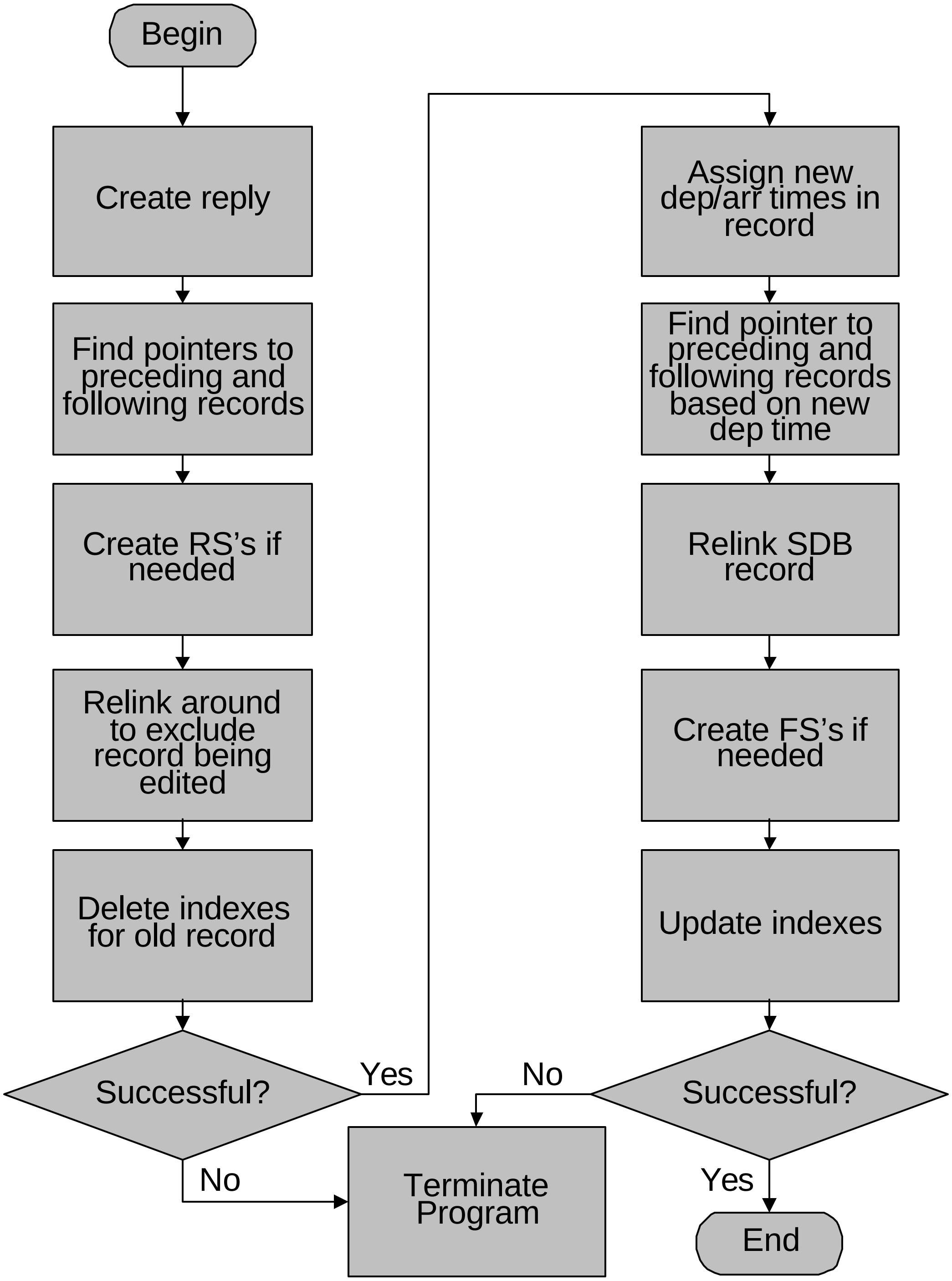

Begin
Create reply
Find pointers to preceding and following records
Create RS's if needed
Relink around to exclude record being edited
Delete indexes for old record
Successful?
Yes
No
Terminate Program
Assign new dep/arr times in record
Find pointer to preceding and following records based on new dep time
Relink SDB record
Create FS's if needed
Update indexes
Successful?
No
Yes
End

Figure 22-9. Logic for the edit_dep_time Routine

Routines associated with FPSD - The following routines are used for the **FPSD** command:

- *sdb_delete_indices* routine has for input the offset in the *SDB* of the new flight being deleted. The routine calls the following five delete indexes routines in turn and then returns a flag indicating whether or not the indexes were deleted successfully:

- *delete_time_table_index* routine has for input the offset in the *SDB* of the flight being deleted. It goes through the time table, relinking all records that pointed to the deleted flight to the next flight (in chronological order) in the time table linked list map file.

- *delete_flight_id_index* routine has for input the offset in the *SDB* of the flight being deleted. The routine first determines if the flight is in the **hash map file**: (1) if it is *not*, the routine returns an error value; (2) if it is, the routine links the deleted flight record as the next available record in the flight ID linked list map file; in addition, if the routine determines that this flight was the only leg for this flight ID in the linked list, it clears the ID from the flight ID **hash map file**.

- *delete_air_carrier_*index routine has for input the offset in the *SDB* of the flight being deleted. The routine first determines if the air carrier is in the **hash map file**: (1) if it is *not*, the routine returns an error value; (2) if it is, the routine links the deleted flight record as the next available record in the air carrier linked list map file; in addition, if the routine determines that this flight was the only flight for this air carrier in the linked list, it clears the air carrier name from the air carrier **hash map file**.

- *delete_arr_airport_index* routine has for input the offset in the *SDB* of the flight being deleted. The routine first determines if the arrival airport is in the **hash map file**: (1) if it is *not*, the routine returns an error value; (2) if it is, the routine links the deleted flight record as the next available record in the arrival airport linked list map file; in addition the routine goes through the time table, relinking all records that pointed to the deleted flight to the next flight (in chronological order) in the arrival airport linked list map file.

- *delete_dep_airport_index* routine has for input the offset in the *SDB* of the flight being deleted. The routine first determines if the departure airport is in the **hash map file**: (1) if it is *not,* the routine returns an error value; (2) if it is, the routine links the deleted flight record as the next available record in the departure airport linked list map file; in addition the routine goes through the time table, relinking all records that pointed to the deleted flight to the next flight (in chronological order) in the departure airport linked list map file.

- *get_hash_air_carrier_ptr* routine returns the pointer in the air carrier **hash map file** to the specified air carrier or to the next empty bucket.

- *get_hash_arr_airport_ptr* routine returns the pointer in the arrival airport **hash map file** to the specified arrival airport or to the next empty bucket.

22.3 The SDB List Server Process

Purpose

The *SDB List Server* interprets **LIST** and **CLIST** requests, creates flight schedule lists in response to them, and formats them for transfer to the *listserver.*

Execution Control

The *SDB List Server* is started along with the *SDB Server* and the *Update/Request Server* by *nodescan*. The *SDB List Server* is invoked with one command line argument; the name of an ASCII file that contains all the necessary initialization parameters, which are described in the Input section.

The *SDB List Server* process runs continuously; if it fails, it is restarted by *nodescan*. The *SDB Server* must be running for the *SDB List Server* to run.

Input

The initialization parameters, read from the arguments file at program startup, are as follows:

- Directory in which the program will run.
- Name of the file with the pathnames of all the database files.
- Name of the TCP/IP file to communicate with the *SDB Server*.
- Name of the file in which the response to a **LIST/CLIST** request will be written.
- Name of the file with aircraft category information.
- Name of the file that associates airports with Air Route Traffic Control Centers (ARTCCs).
- Name of the element pair file.
- Name of the file with airline categories.

After initialization, the inputs (consisting of messages from the *SDB Server)* are as follows:

- **LIST/CLIST** requests.
- **UNMAP** commands.

- **MAP** commands.
- **NOOP** messages.

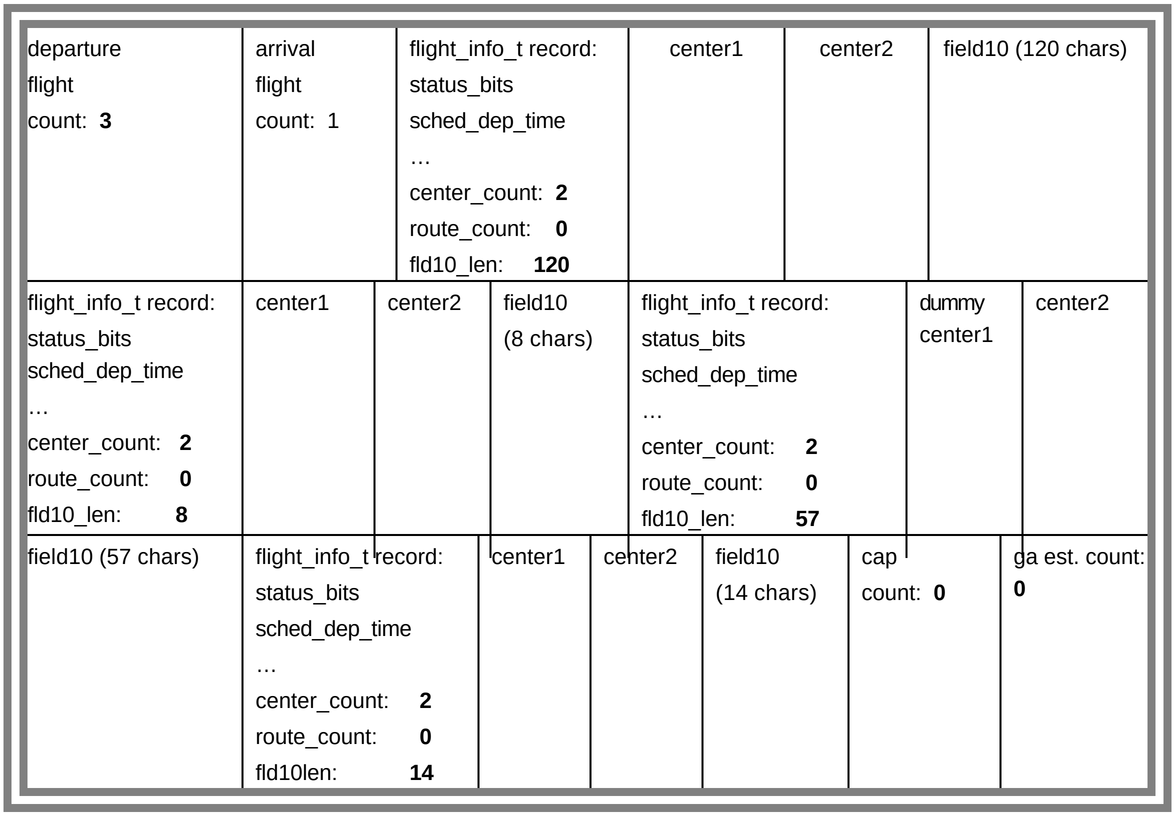
Output

The response to the **LIST/CLIST** request consists of the flight schedules in the format of the flight_info_t data structure shared by the *SDB List Server* and the *listserver*. These **flight_info_t** records are put in the file named in the fourth input parameter. A message is returned to the *SDB Server* that indicates the name of the file or, in case of an error, a specific error message.

- File that contains the flight schedules in response to a **LIST** request
- Messages sent through the internal channel to the *SDB Server*

The **LIST** request response file is organized as shown in Figure 22-10. The first record consists of two short integers: the first is the count of *departure* flight records of **type flight _info_t** (see Table 22-2 in Section 22.5) that immediately follow the second integer. The second integer is the count of *arrival* flight records. Next is the first **flight_info_t** record, which contains count fields for the following items:

- Fixes
- Waypoints
- Sectors
- ARTCCs
- Airways



departure flight count: **3**	arrival flight count: 1	flight_info_t record: status_bits sched_dep_time … center_count: **2** route_count: **0** fld10_len: **120**	center1	center2	field10 (120 chars)

flight_info_t record: status_bits sched_dep_time … center_count: **2** route_count: **0** fld10_len: **8**	center1	center2	field10 (8 chars)	flight_info_t record: status_bits sched_dep_time … center_count: **2** route_count: **0** fld10_len: **57**	dummy center1	center2

field10 (57 chars)	flight_info_t record: status_bits sched_dep_time … center_count: **2** route_count: **0** fld10len: **14**	center1	center2	field10 (14 chars)	cap count: **0**	ga est. count: **0**

Figure 22-10. Format of the LIST Response File

At this time, the counts are zero for all of the elements but the ARTCCs. After the first **flight_info_t** record are the ARTCCs; the number of ARTCCs following the **flight_info_t** record is specified in the **counts** fields for that item. Each ARTCC is represented by a single-character code. If there is only one ARTCC, the ARTCC count is raised to two and a dummy ARTCC (a single-space character) is filled in so the *arrival* ARTCC and *departure* ARTCC can be distinguished by the *listserver*. The final field in the **flight_info_t** record contains the length of the field10 text. The field10 text is placed immediately after all of the element names.

There are **flight_info_t** records (with their accompanying elements) that correspond to the total number of departures and arrivals shown in the beginning of the file. Following the **flight_info_t** records are counts and records for capacity and general aviation (GA) estimates data (see Table 22-2 in Section 22.5): (1) a count shows the number of capacity values that follow and the **capga_info_t** records containing those values; (2) a count shows the number of GA estimates that follow and the **capga_info_t** records with that data. Currently, these counts are always zeros.

Processing

During program startup, the *SDB List Server* opens and reads the input parameters file (named in the command line argument to the program invocation). It reads the parameters described in the Input section. Initialization consists of moving to the directory named in the first input parameter; mapping the files containing the *SDB* (the names of which are found in the file named in the second input parameter); opening a channel to the *SDB Server*; and setting up an event counter for messages received in, or taken out of, that channel. Initialization is completed by mapping the files named in the remaining input parameters.

After initialization, the *SDB List Server* goes to sleep until an event is triggered. When an event occurs, it is processed according to its type. The possible events are as follows:

- Internal *Schedule Database* function message received **-** a message sent from the *SDB Server*. It is one of three commands:

  (1) **LIST** or **CLIST** command received by the *SDB Server* from the *ETMS Communications* functions.

  (2) **UNMAP** command generated by the *SDB Server.* This command is sent when the *SDB Server* receives a request it must pass on to the *Update/Request Server*. The *SDB List Server* must unmap the *SDB* files to allow the *Update/Request Server* to access them for writing. After unmapping the files, the *SDB List Server* returns an acknowledgement to the *SDB Server*, so it can send the original request on to the *Update/Request Server*.

  (3) **MAP** command generated by the *SDB Server*. This command is sent after the *Update/Request Server* has finished processing a request. The *SDB List Server* can now remap the *SDB* files. After remapping the files, the *SDB List Server* returns an acknowledgement to the *SDB Server*.

- Message removed from internal *SDB* channel **-** indicates room may be available in the channel for a message that could not be sent earlier when the channel was full. This event is important only when attempts to send messages to the *SDB* Server fail because the channel is full. An event of this type signals the program that a message may have been removed from the channel, making room for more messages. The most recent message is then put into the channel.

- System timer event triggered **-** a system timer set up in initialization, triggered every few minutes. The exact number of minutes is determined by the constant COM_SERVER_NOOP_TIME. The latter is a system-wide value determined by the frequency that the *ETMS Communications* functions issue a **NOOP** message to all connected channels.

> An equivalent timer event in the *SDB Server* process triggers the sending of a **NOOP** message from the *SDB Server* to the *SDB List Server*; the *SDB List Server* uses this message to check the status of its connection to the *SDB Server*. If the *SDB List Server* receives *fewer* **NOOP** messages than expected, the program assumes it has lost its connection to the *SDB Server* and tries to reconnect.

The main function of the *SDB List Server* is to respond to **LIST** and **CLIST** requests. The request is parsed first. **LIST** and **CLIST** requests are both handled in the same way by the *SDB List Server*. In the parsing process, any token in the **location identifier** position preceded by an asterisk is treated as if it were an ARTCC code, and expanded into a list of all the airports in that ARTCC. Location identifiers *not* preceded by an asterisk are assumed to be airport names. Each airport name in the request is stored in an array of airport names. The list response file name in the fourth input parameter is created. A departure flights count value of **zero** is put in the first two bytes of the file; and an arrival flights count value of zero is put in the second two bytes (see Figure 22-10) to hold the place for the actual flight counts, written in when the total number of departure and arrival flights is known.

A linked list is created for each airport in the airport names array of all flights in the database either departing or arriving from that airport between the start and end times specified in the request. Each flight on the list is checked to ensure that it flies on the date specified in the request. Both days of the week and the effective/discontinue dates in the *SDB* record of the flight are examined to make this determination. If the flight does occur, information from its *SDB* record is used to fill the fields of a **flight_info_t** record (see Table 22-2). The **flight_info_t** record is written to the **list response** file. Immediately after, any ARTCC codes and the **field10** are written into the file. A separate count is kept of all departure and arrival flights entered into the file, and those counts are written to the first four bytes of the file after all the flights have been entered. After the file is closed, a message containing the name of the list response file is returned to the *SDB Server*.

Error Conditions and Handling

The *SDB List Server* writes all its diagnostic messages to the *listserver.pad.timestamp* window. Errors occurring during initialization are all fatal. An appropriate error message is written to the screen, and the program exits. The *listserver.pad.timestamp* is saved: therefore, any messages written after the pad has successfully opened will be saved for examination. The error messages that can appear during initialization are the following:

- No filename passed in as argument.
- Can't open arguments file.
- Can't map *SDB* files.
- No TCP/IP file name in arguments file.

- No name for **LIST/CLIST** output file.
- No aircraft categories filename in args file.
- Can't map aircraft categories file.
- Unable to initialize airport pairs database.
- No element-in-center filename in args file.
- Can't initialize element-by-center table.
- No element-pair filename in args file.

The *SDB List Server* generates a number of **LIST**-request-specific error messages passed back to the *SDB Server*. These are listed in Section 22.1 under *Error Conditions and Handling*. The error messages that are passed back are also written to the *listserver.pad.timestamp*, prefaced by the name of the routine in which the error occurred. Errors that occur in system calls are written to the process pad.

22.4 Schedule Database Function Source Code Organization

The *Schedule Database* function source code resides in C files under configuration management using ClearCase.

22.5 Schedule Database Function Data Structures

Tables 22-1 through 22-4 describe the data structures used by *SDB*.

Table 22-1. Node Data Structure

node				
Library Name: sdb_openlib		**Purpose:** element of *updates* or *requests* queues		
Element Name: sdb_upreq_header.h				
Data Item	**Definition**	**Unit/Format**	**Range**	**Var. Type/Bits**
mes	buffer containing FS, RS, or request messages			up_m_t
mesnum	number of messages in the buff- er		up to MAX	short
ptype	processtype – Update or Request	single character	U or R	char
fst_tstamp	timestamp of first FS message in buffer			CALTIME
lst_tstamp	timestamp of last FS message in buffer			CALTIME

prev	pointer to previous queue node			npt
next	pointer to next queue node			npt

Table 22-2. flight_info_t Data Structure

flight_info_t

Library Name: sdb_openlib	**Purpose**: To format data for retrieval by the *listserver*
Element Name: sdb_list_interface.h	
Field Name: status_bits	**Field Type:** short

Data Item	Definition	Unit/Format	Range	Var. Type/Bits
taxi_bit	is this a taxi flight?	1 = yes, 0 = no	0 - 1	bit 15
cancel_bit	was this flight canceled (by a CXSD command)?	1 = yes, 0 = no	0 – 1	bit 14
add_bit	was this flight added (by an FPSD command)?	1 = yes, 0 = no	0 – 1	bit 13
delete_bit	was this flight deleted (not currently being used)?	1 = yes, 0 = no	0 – 1	bit 12
inhibit_bit	was this flight inhibited (by an INHB command)?	1 = yes, 0 = no	0 - 1	bit 11
sched_dep_time	scheduled departure time	julian date/time	-	INT32
sched_arr_time	scheduled arrival time	julian date/time	-	INT32
flight_id	flight id	text	-	string7
dep_ap	departure airport	text	-	string5
arr_ap	arrival airport	text	-	string5
	estimated time en route	minutes	1 – 1440	short
ac_cat_code	aircraft category code	single character	J, T, or P	char
user_category	airline category code	single character		char
ac_name	aircraft name	text	-	string4
fix_count	number of fix names to follow		-	short
waypt_count	number of waypoints to follow		-	short
sector_count	number of sectors to follow		-	short
center_count	number of ARTCC codes to follow		-	short
route_count	number of routes to follow		-	short
fld10_len	length of field10 to follow		0 – 255	short

Table 22-3. capga_info_t Data Structure

capga_info_t	
Library Name: sdb_openlib	**Purpose:** To format Capacity or GA Estimates data for retrieval by the *listserver*
Element Name: sdb_list_interface_h	

Data Item	Definition	Unit/Format	Range	Var. Type/Bits
ap	airport name	two-to four- letter code	-	string5
starttime	beginning of time range	hours and minutes	0-2359	short
endtime	end of time range	hours and minutes	0-2359	short
arrivals	number of arrivals	flights per 15 minutes	-	short
departures	number of departures	flights per 15 minutes	-	short

Table 22-4. q_t Data Structure

q_t	
Library Name: sdb_openlib	**Purpose:** q_t is the queue structure used in *updates* and *requests* queue to store FS buffers and requests from SDB_server
Element Name: sdb_upreq_header_h	

Data Item	Definition	Unit/Format	Range	Var. Type/Bits
first	pointer to the first node in the queue			npt
last	pointer to the last node in the queue			npt
max	maximum number of nodes allowable in the queue		based on parameter	short
qnum	number of nodes in the queue			short

high	unusually high but still allowable number of nodes for warnings		based on parameter	short